First Depth First Search

You are given N towns (1 to N). All towns are connected via unique directed path as mentioned in the input.

Given 2 towns find whether you can reach the first town from the second without repeating any edge.

B C : query to find whether B is reachable from C.

Input contains an integer array **A** of size N and 2 integers B and C (1 <= B, C <= N).

There exist a directed edge from A[i] to i+1 for every 1 <= i < N. Also, it's guaranteed that A[i] <= i for every 1 <= i < N.

NOTE: Array A is 0-indexed. A[0] = 1 which you can ignore as it doesn't represent any edge.

Problem Constraints

1 <= **N** <= 100000

Input Format

First argument is vector **A**

Second argument is integer **B**

Third argument is integer **C**

Output Format

Return 1 if reachable, 0 otherwise.

Example Input

Input 1:

```
A = [1, 1, 2]
 B = 1
 C = 2
```

Input 2:

```
A = [1, 1, 2]
 B = 2
 C = 1
```

Example Output

Output 1:

```
0
```

Output 2:

```
1
```

Example Explanation

Explanation 1:

```
Tree is 1--> 2--> 3 and hence 1 is not reachable from 2.
```

Explanation 2:

```
Tree is 1--> 2--> 3 and hence 2 is reachable from 1.
```

```
public class Solution {
    // Adjacency list representation of the graph
    private List<List<Integer>> adj;
    // Visited array to keep track of visited nodes during DFS
    private boolean[] visited;

    public int solve(int[] A, final int B, final int C) {
        int N = A.length;

        // Initialize the adjacency list
        adj = new ArrayList<>();
        for (int i = 0; i <= N; i++) {
            adj.add(new ArrayList<>());
        }

        // Build the graph based on the input array A
        for (int i = 1; i < N; i++) {
            adj.get(A[i]).add(i + 1); // Add a directed edge from A[i] to i+1
        }

        // Initialize the visited array
        visited = new boolean[N + 1];
```

```
        // Perform DFS from node C to check if B is reachable
        return dfs(C, B) ? 1 : 0;
    }

    private boolean dfs(int current, int target) {
        if (current == target) {
            return true; // Reached the target node
        }

        visited[current] = true; // Mark the current node as visited

        // Recur for all the vertices adjacent to this vertex
        for (int neighbor : adj.get(current)) {
            if (!visited[neighbor]) {
                if (dfs(neighbor, target)) {
                    return true; // Path found to target node
                }
            }
        }

        return false; // No path found from this node
    }}
```

Number of islands

Given a matrix of integers A of size N x M consisting of 0 and 1. A group of connected 1's forms an island. From a cell (i, j) such that A[i][j] = 1 you can visit any cell that shares a corner with (i, j) and value in that cell is 1.

More formally, from any cell (i, j) if A[i][j] = 1 you can visit:

(i-1, j) if (i-1, j) is inside the matrix and A[i-1][j] = 1.
(i, j-1) if (i, j-1) is inside the matrix and A[i][j-1] = 1.
(i+1, j) if (i+1, j) is inside the matrix and A[i+1][j] = 1.
(i, j+1) if (i, j+1) is inside the matrix and A[i][j+1] = 1.
(i-1, j-1) if (i-1, j-1) is inside the matrix and A[i-1][j-1] = 1.
(i+1, j+1) if (i+1, j+1) is inside the matrix and A[i+1][j+1] = 1.
(i-1, j+1) if (i-1, j+1) is inside the matrix and A[i-1][j+1] = 1.
(i+1, j-1) if (i+1, j-1) is inside the matrix and A[i+1][j-1] = 1.

Return the number of islands.

NOTE: Rows are numbered from top to bottom and columns are numbered from left to right.

```
import java.util.*;
public class Solution {
    public int solve(ArrayList<ArrayList<Integer>> A) {
        if (A == null || A.isEmpty()) {
            return 0;
        }

        int N = A.size(); // Number of rows
        int M = A.get(0).size(); // Number of columns
        boolean[][] visited = new boolean[N][M]; // Array to keep track of visited cells
        int islandsCount = 0; // Count of islands

        // Traverse each cell in the matrix
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                if (A.get(i).get(j) == 1 && !visited[i][j]) {
                    // Found an unvisited island cell
                    islandsCount++;
                    // Perform DFS to mark all connected '1's as visited
                    dfs(A, visited, i, j);
                }
            }
        }

        return islandsCount;
    }

    // Depth-First Search (DFS) to mark all connected '1's as visited
    private void dfs(ArrayList<ArrayList<Integer>> A, boolean[][] visited, int row, int col)
{
```

```
        // Directions arrays for moving up, down, left, right, and diagonals
        int[] rowDir = {-1, 1, 0, 0, -1, -1, 1, 1};
        int[] colDir = {0, 0, -1, 1, -1, 1, -1, 1};

        visited[row][col] = true; // Mark current cell as visited

        // Explore all 8 possible directions
        for (int k = 0; k < 8; k++) {
            int newRow = row + rowDir[k];
            int newCol = col + colDir[k];

            // Check if the new cell is within bounds and is a valid island cell
            if (isValid(A, visited, newRow, newCol)) {
                dfs(A, visited, newRow, newCol); // Recursively call DFS
            }
        }
    }

    // Helper method to check if a cell is valid and can be visited
    private boolean isValid(ArrayList<ArrayList<Integer>> A, boolean[][] visited, int row,
int col) {
        int N = A.size();
        int M = A.get(0).size();

        // Check if row and column are within bounds and cell value is '1' and not visited
        return (row >= 0 && row < N && col >= 0 && col < M && A.get(row).get(col) == 1
&& !visited[row][col]);
    }
}
```

Cycle in Directed Graph

Given an directed graph having **A** nodes. A matrix **B** of size `M x 2` is given which represents the **M** edges such that there is a edge directed from node **B[i][0]** to node **B[i][1]**.

Find whether the graph contains a cycle or not, return **1** if cycle is present else return **0**.

NOTE:

The cycle must contain atleast two nodes.
There are no self-loops in the graph.
There are no multiple edges between two nodes.
The graph may or may not be connected.
Nodes are numbered from 1 to A.
Your solution will run on multiple test cases. If you are using global variables make sure to clear them.

```
public class Solution {
    // Adjacency list representation of the graph
    private List<List<Integer>> adj;
    // Color array to keep track of the state of each node
    private int[] color;

    // Constants representing the state of each node
    private static final int WHITE = 0; // Node has not been visited
    private static final int GRAY = 1;  // Node is in the current DFS path
    private static final int BLACK = 2; // Node has been fully processed

    public int solve(int A, int[][] B) {
        // Initialize the adjacency list
        adj = new ArrayList<>();
        for (int i = 0; i <= A; i++) {
            adj.add(new ArrayList<>());
        }

        // Build the graph
        for (int[] edge : B) {
            adj.get(edge[0]).add(edge[1]);
```

```
    }

    // Initialize the color array
    color = new int[A + 1];

    // Check for cycles in each connected component
    for (int i = 1; i <= A; i++) {
        if (color[i] == WHITE) {
            if (dfs(i)) {
                return 1; // Cycle found
            }
        }
    }

    return 0; // No cycle found
}

private boolean dfs(int node) {
    // Mark the current node as being in the current DFS path (GRAY)
    color[node] = GRAY;

    // Recur for all the vertices adjacent to this vertex
    for (int neighbor : adj.get(node)) {
        if (color[neighbor] == WHITE) {
            if (dfs(neighbor)) {
                return true;
            }
        } else if (color[neighbor] == GRAY) {
            return true; // Cycle detected
        }
    }

    // Mark the node as fully processed (BLACK)
    color[node] = BLACK;
```

```
        return false;
    }
}
```

Path in Directed Graph

Problem Description

Given an directed graph having **A** nodes labelled from **1** to **A** containing **M** edges given by matrix **B** of size `M x 2`such that there is a edge directed from node

B[i][0] to node **B[i][1]**.

Find whether a path exists from node **1** to node **A**.

Return **1** if path exists else return **0**.

NOTE:

There are no self-loops in the graph.
There are no multiple edges between two nodes.
The graph may or may not be connected.
Nodes are numbered from 1 to A.
Your solution will run on multiple test cases. If you are using global variables make sure to clear them.

```
public class Solution {
    // Adjacency list representation of the graph
    private List<List<Integer>> adj;
    // Visited array to keep track of visited nodes during DFS
    private boolean[] visited;

    public int solve(int A, int[][] B) {
        // Initialize the adjacency list
        adj = new ArrayList<>();
        for (int i = 0; i <= A; i++) {
            adj.add(new ArrayList<>());
        }
```

```
    // Build the graph
    for (int[] edge : B) {
        adj.get(edge[0]).add(edge[1]);
    }

    // Initialize the visited array
    visited = new boolean[A + 1];

    // Perform DFS from node 1
    if (dfs(1, A)) {
        return 1; // Path exists
    } else {
        return 0; // Path does not exist
    }
}

private boolean dfs(int node, int target) {
    if (node == target) {
        return true; // Reached the target node
    }

    visited[node] = true; // Mark the current node as visited

    // Recur for all the vertices adjacent to this vertex
    for (int neighbor : adj.get(node)) {
        if (!visited[neighbor]) {
            if (dfs(neighbor, target)) {
                return true; // Path found to target node
            }
        }
    }

    return false; // No path found from this node
}
```

```
}
```

Rotten Oranges

Problem Description

Given a matrix of integers **A** of size N x M consisting of 0, 1 or 2.

Each cell can have three values:

The value 0 representing an empty cell.

The value 1 representing a fresh orange.

The value 2 representing a rotten orange.

Every minute, any fresh orange that is adjacent (Left, Right, Top, or Bottom) to a rotten orange becomes rotten. Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1 instead.

Note: Your solution will run on multiple test cases. If you are using global variables, make sure to clear them.

```
import java.util.LinkedList;
import java.util.Queue;

// Custom class to store row, column, and time in the queue
class Orange {
    int row;
    int col;
    int time;

    public Orange(int row, int col, int time) {
        this.row = row;
        this.col = col;
        this.time = time;
    }
}
```

```
public class Solution {
    // Directions arrays for moving in four possible directions (right, down, left, up)
    private static final int[] xdir = {0, 1, 0, -1};
    private static final int[] ydir = {1, 0, -1, 0};

    public int solve(int[][] A) {
        int rows = A.length;
        int cols = A[0].length;
        Queue<Orange> queue = new LinkedList<>();
        int freshOranges = 0;

        // Initialize the queue with all initially rotten oranges and count fresh oranges
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                if (A[r][c] == 2) {
                    queue.add(new Orange(r, c, 0)); // Time is initially 0 for all rotten oranges
                } else if (A[r][c] == 1) {
                    freshOranges++;
                }
            }
        }

        // If there are no fresh oranges initially, return 0
        if (freshOranges == 0) return 0;

        int minutes = 0;

        // BFS traversal
        while (!queue.isEmpty()) {
            Orange orange = queue.poll();
            int row = orange.row;
            int col = orange.col;
            int time = orange.time;
```

```
            // Try to rot adjacent fresh oranges
            for (int d = 0; d < 4; d++) {
                int newRow = row + xdir[d];
                int newCol = col + ydir[d];

                if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols &&
A[newRow][newCol] == 1) {
                    A[newRow][newCol] = 2; // Rot this orange
                    queue.add(new Orange(newRow, newCol, time + 1)); // Increment time for
rotted oranges
                    freshOranges--;
                }
            }
            // Update minutes to the maximum time found in the queue
            minutes = time;
        }
        // If there are still fresh oranges left, return -1
        return freshOranges == 0 ? minutes : -1;
    }
}
```

Another BFS

Problem Description

Given a weighted undirected graph having A nodes, a source node C and destination node D.

Find the shortest distance from C to D and if it is impossible to reach node D from C then return -1.

You are expected to do it in Time Complexity of O(A + M).

Note:

There are no self-loops in the graph.

No multiple edges between two pair of vertices.

The graph may or may not be connected.

Nodes are Numbered from 0 to A-1.

Your solution will run on multiple testcases. If you are using global variables make sure to clear them.

```
import java.util.*;

public class Solution {
    // Maximum number of nodes in the graph
    static int MAX_NODES = 200009;

    // Array to track visited nodes
    static int[] visited = new int[MAX_NODES];

    // Adjacency list representation of the graph
    static ArrayList<ArrayList<Integer>> adjList;

    // Method to initialize the graph
    public static void initializeGraph() {
        adjList = new ArrayList<ArrayList<Integer>>(MAX_NODES);
        for (int i = 0; i < MAX_NODES; i++) {
            visited[i] = 0; // Initialize all nodes as unvisited
            adjList.add(new ArrayList<Integer>());
        }
    }

    // Method to solve the problem
    public int solve(int nodes, int[][] edges, int source, int destination) {
        initializeGraph(); // Initialize the graph

        // Build the graph from the given edges
        for (int[] edge : edges) {
            int u = edge[0];
```

```
            int v = edge[1];
            int weight = edge[2];

            // Regular edge (weight == 1) or weighted edge (weight == 2)
            if (weight == 1) {
                adjList.get(u).add(v); // Add edge between u and v
                adjList.get(v).add(u); // Add edge between v and u (undirected graph)
            } else {
                adjList.get(u).add(u + nodes); // Add dummy node for u
                adjList.get(u + nodes).add(v); // Connect dummy node of u to v
                adjList.get(v).add(v + nodes); // Add dummy node for v
                adjList.get(v + nodes).add(u); // Connect dummy node of v to u
            }
        }

        // Call BFS to find the shortest path distance
        return bfs(source, destination);
    }

    // Method to perform BFS and find the shortest path distance
    public static int bfs(int start, int end) {
        visited[start] = 1; // Mark the start node as visited
        Queue<Pair> queue = new ArrayDeque<Pair>();
        queue.offer(new Pair(start, 0)); // Add the start node to the queue with distance 0

        while (!queue.isEmpty()) {
            Pair p = queue.poll();
            int node = p.first; // Current node
            int dist = p.second; // Distance from start node

            // If we reach the end node, return the distance
            if (node == end)
                return dist;
```

```
            // Explore neighbors of the current node
            for (int neighbor : adjList.get(node)) {
                if (visited[neighbor] == 0) {
                    visited[neighbor] = 1; // Mark the neighbor as visited
                    queue.offer(new Pair(neighbor, dist + 1)); // Add neighbor to the queue with
updated distance
                }
            }
        }

        return -1; // If end node is unreachable, return -1
    }
}

// Pair class to store a pair of integers
class Pair {
    int first; // First element of the pair
    int second; // Second element of the pair

    // Constructor to initialize the pair
    public Pair(int a, int b) {
        this.first = a;
        this.second = b;
    }
}
```

Check Bipartite Graph

Problem Description

Given a undirected graph having **A** nodes. A matrix **B** of size M x 2 is given which represents the edges such that there is an edge between B[i][0] and B[i][1].

Find whether the given graph is bipartite or not.

A graph is bipartite if we can split it's set of nodes into two independent subsets A and B such that every edge in the graph has one node in A and another node in B

Note:

There are no self-loops in the graph.

No multiple edges between two pair of vertices.

The graph may or may not be connected.

Nodes are Numbered from 0 to A-1.

Your solution will run on multiple testcases. If you are using global variables make sure to clear them.

```
import java.util.*;

public class Solution {
    // Adjacency list representation of the graph
    private ArrayList<ArrayList<Integer>> adj;

    public int solve(int A, int[][] B) {
        adj = new ArrayList<>();
        for (int i = 0; i < A; i++) {
            adj.add(new ArrayList<>());
        }

        // Build the adjacency list from the edge list
        for (int[] edge : B) {
            int u = edge[0];
            int v = edge[1];
            adj.get(u).add(v);
            adj.get(v).add(u); // Since it's an undirected graph
        }

        // Array to store the color of each node
        int[] color = new int[A];
        Arrays.fill(color, -1); // -1 means uncolored
```

```
        // Check bipartiteness for each component (in case the graph is not connected)
        for (int i = 0; i < A; i++) {
            if (color[i] == -1) { // If the node is not colored
                if (!bfs(i, color)) {
                    return 0; // Not bipartite
                }
            }
        }
             return 1; // Bipartite
    }

    // BFS to check bipartiteness
    private boolean bfs(int start, int[] color) {
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(start);
        color[start] = 0; // Start coloring with 0

        while (!queue.isEmpty()) {
            int node = queue.poll();
            int currentColor = color[node];
            int neighborColor = 1 - currentColor; // Alternate color

            for (int neighbor : adj.get(node)) {
                if (color[neighbor] == -1) {
                    color[neighbor] = neighborColor;
                    queue.offer(neighbor);
                } else if (color[neighbor] != neighborColor) {
                    return false; // Found a conflict, not bipartite
                }
            }
        }
             return true; // Graph is bipartite
    }
}
```

Commutable Islands

Problem Description

There are **A** islands and there are **M** bridges connecting them. Each bridge has some **cost** attached to it.

We need to find bridges with **minimal cost** such that all islands are connected.

It is guaranteed that input data will contain **at least one** possible scenario in which all islands are connected with each other.

```
public class Solution {
    static class Edge implements Comparable<Edge> {
        int to;
        int cost;

        public Edge(int to, int cost) {
            this.to = to;
            this.cost = cost;
        }

        @Override
        public int compareTo(Edge other) {
            return Integer.compare(this.cost, other.cost);
        }
    }

    public int solve(int A, int[][] B) {
        // Step 1: Build the adjacency list
        List<List<Edge>> adj = new ArrayList<>();
        for (int i = 0; i < A; i++) {
            adj.add(new ArrayList<>());
        }

        for (int[] bridge : B) {
            int from = bridge[0] - 1; // Convert 1-based to 0-based index
```

```
            int to = bridge[1] - 1;   // Convert 1-based to 0-based index
            int cost = bridge[2];
            adj.get(from).add(new Edge(to, cost));
            adj.get(to).add(new Edge(from, cost));
        }

        // Step 2: Use Prim's algorithm to find MST
        int minCost = primMST(adj, A);

        return minCost;
    }

    private int primMST(List<List<Edge>> adj, int A) {
        // Priority queue to store the minimum cost edges
        PriorityQueue<Edge> pq = new PriorityQueue<>();
        // Visited array to mark visited islands
        boolean[] visited = new boolean[A];
        // Start from island 0 (arbitrary choice)
        pq.offer(new Edge(0, 0));
        int minCost = 0;
        int edgesUsed = 0;

        while (!pq.isEmpty()) {
            Edge current = pq.poll();
            int island = current.to;
            int cost = current.cost;

            if (visited[island]) {
                continue;
            }

            // Add cost of the edge to the MST
            minCost += cost;
            visited[island] = true;
```

```
            edgesUsed++;

            // If we have added A-1 edges, we are done
            if (edgesUsed == A) {
                break;
            }

            // Add all adjacent edges of the current island to the priority queue
            for (Edge neighbor : adj.get(island)) {
                if (!visited[neighbor.to]) {
                    pq.offer(neighbor);
                }
            }
        }

        return minCost;
    }
}
```

Dijkstra

Problem Description

Given a **weighted undirected graph** having **A nodes** and **M weighted edges**, and a **source node C**.

You have to **find an integer array D of size A** such that:

D[i]: Shortest distance from the **C node** to **node i**.
If **node i** is not reachable from C then **-1**.

Note:

There are no self-loops in the graph.
There are no multiple edges between two pairs of vertices.
The graph may or may not be connected.
Nodes are numbered from 0 to A-1.

Your solution will run on multiple test cases. If you are using global variables, make sure to clear them.

```
import java.util.*;
public class Solution {
    public int[] solve(int A, int[][] B, int C) {
        List<List<Node>> adjList = new ArrayList<>();
        for (int i = 0; i < A; i++) {
            adjList.add(new ArrayList<>());
        }

        // Build the adjacency list
        for (int[] edge : B) {
            int u = edge[0];
            int v = edge[1];
            int weight = edge[2];
            adjList.get(u).add(new Node(v, weight));
            adjList.get(v).add(new Node(u, weight)); // Since it's an undirected graph
        }

        // Distance array to store shortest distances from source C
        int[] distances = new int[A];
        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[C] = 0;

        // Min-heap priority queue for Dijkstra's algorithm
        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(node
-> node.distance));
        pq.offer(new Node(C, 0));

        while (!pq.isEmpty()) {
            Node currNode = pq.poll();
            int u = currNode.vertex;
            int distU = currNode.distance;
```

```
        // If distU is already greater than the known shortest path, skip it
        if (distU > distances[u]) {
            continue;
        }

        // Traverse all neighbors of u
        for (Node neighbor : adjList.get(u)) {
            int v = neighbor.vertex;
            int weightUV = neighbor.distance;

            // Calculate the new distance
            int newDist = distU + weightUV;

            // If a shorter path to v is found, update distances and push to pq
            if (newDist < distances[v]) {
                distances[v] = newDist;
                pq.offer(new Node(v, newDist));
            }
        }
    }

    // Replace unreachable nodes (still with Integer.MAX_VALUE) with -1
    for (int i = 0; i < A; i++) {
        if (distances[i] == Integer.MAX_VALUE && i != C) {
            distances[i] = -1;
        }
    }

    return distances;
}

// Node class to represent vertices and distances in the graph
static class Node {
    int vertex;
```

```
        int distance;

        public Node(int vertex, int distance) {
            this.vertex = vertex;
            this.distance = distance;
        }
    }
}
```

Topological Sort

Problem Description

Given an directed acyclic graph having **A** nodes. A matrix **B** of size `M x 2` is given which represents the **M** edges such that there is a edge directed from node **B[i][0]** to node **B[i][1]**.

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Return the **topological ordering** of the graph and if it doesn't exist then return an empty array.

If there is a solution return the correct ordering. If there are multiple solutions print the **lexographically smallest one**.

Ordering (a, b, c) is said to be lexographically smaller than ordering (e, f, g) if a < e or if(a==e) then b < f and so on.

NOTE:

There are no self-loops in the graph.
The graph may or may not be connected.
Nodes are numbered from 1 to A.
Your solution will run on multiple test cases. If you are using global variables make sure to clear them.

```
import java.util.*;
```

```
public class Solution {
    public int[] solve(int A, int[][] B) {
        List<List<Integer>> adj = new ArrayList<>();
        int[] inDegree = new int[A + 1];

        // Initialize adjacency list and in-degrees
        for (int i = 0; i <= A; i++) {
            adj.add(new ArrayList<>());
        }

        for (int[] edge : B) {
            int u = edge[0];
            int v = edge[1];
            adj.get(u).add(v);
            inDegree[v]++;
        }

        // Priority queue to store nodes with the smallest lexicographical order
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int i = 1; i <= A; i++) {
            if (inDegree[i] == 0) {
                pq.offer(i);
            }
        }

        List<Integer> result = new ArrayList<>();
        while (!pq.isEmpty()) {
            int u = pq.poll();
            result.add(u);

            for (int v : adj.get(u)) {
                inDegree[v]--;
                if (inDegree[v] == 0) {
                    pq.offer(v);
```

```
            }
        }
    }

    // If the result size is not equal to A, there is a cycle
    if (result.size() != A) {
        return new int[0]; // Return empty array
    }

    // Convert list to array
    int[] resArray = new int[A];
    for (int i = 0; i < A; i++) {
        resArray[i] = result.get(i);
    }
    return resArray;
  }
}
```

Possibility of Finishing

Problem Description

There are a total of **A** courses you have to take, labeled from **1** to **A**.

Some courses may have prerequisites, for example to take course **2** you have to first take course **1**, which is expressed as a pair: **[1,2]**.

So you are given two integer array **B** and **C** of same size where for each i **(B[i], C[i])** denotes a pair.

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

Return **1** if it is **possible** to finish all the courses, or **0** if it is **not possible** to finish all the courses.

Problem Constraints

$1 <= A <= 6*10^4$

1 <= length(B) = length(C) <= 10^5

1 <= B[i], C[i] <= A

Input Format

The first argument of input contains an integer A, representing the number of courses.

The second argument of input contains an integer array, B.

The third argument of input contains an integer array, C.

Output Format

Return **1** if it is **possible** to finish all the courses, or **0** if it is **not possible** to finish all the courses.

Example Input

Input 1:

```
A = 3
 B = [1, 2]
 C = [2, 3]
```

Input 2:

```
A = 2
 B = [1, 2]
 C = [2, 1]
```

Example Output

Output 1:

```
1
```

Output 2:

```
0
```

Example Explanation

Explanation 1:

```
It is possible to complete the courses in the following order:
    1 -> 2 -> 3
```

Explanation 2:

```
It is not possible to complete all the courses.
```

```
import java.util.*;

public class Solution {
    public int solve(int A, int[] B, int[] C) {
        // Initialize adjacency list and in-degrees
        List<List<Integer>> adj = new ArrayList<>();
        int[] inDegree = new int[A + 1];
        for (int i = 0; i <= A; i++) {
            adj.add(new ArrayList<>());
        }

        // Build the graph and calculate in-degrees
        for (int i = 0; i < B.length; i++) {
            adj.get(B[i]).add(C[i]);
            inDegree[C[i]]++;
        }

        // Perform topological sort using Kahn's algorithm
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 1; i <= A; i++) {
```

```
            if (inDegree[i] == 0) {
                queue.offer(i);
            }
        }

        int count = 0;
        while (!queue.isEmpty()) {
            int course = queue.poll();
            count++;
            for (int nextCourse : adj.get(course)) {
                inDegree[nextCourse]--;
                if (inDegree[nextCourse] == 0) {
                    queue.offer(nextCourse);
                }
            }
        }

        // If all courses are successfully completed, return 1
        return count == A ? 1 : 0;
    }
}
```

Clone Graph

Problem Description

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

Note: The test cases are generated in the following format (use the following format to use **See Expected Output** option):
First integer N is the number of nodes.
Then, N intgers follow denoting the label (or hash) of the N nodes.
Then, N^2 integers following denoting the adjacency matrix of a graph, where **Adj[i][j] = 1** denotes presence of an undirected edge between the i^{th} and j^{th} node, **0** otherwise.

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     List<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) { label = x; neighbors = new
ArrayList<UndirectedGraphNode>(); }
 * };
 */

class UndirectedGraphNode {
    int label;
    List<UndirectedGraphNode> neighbors;

    UndirectedGraphNode(int x) {
        label = x;
        neighbors = new ArrayList<>();
    }
}

public class Solution {
    // HashMap to store original node -> cloned node mapping
    HashMap<UndirectedGraphNode, UndirectedGraphNode> map;

    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        // Initialize the map
        map = new HashMap<>();
        // Call the helper function to clone the graph
        return auxCloneGraph(node);
    }

    // Helper function to perform the cloning using DFS
    public UndirectedGraphNode auxCloneGraph(UndirectedGraphNode node) {
        // If the current node is null, return null
```

```
        if (node == null)F
            return node;

        // If the node is already cloned, return its clone from the map
        if (map.containsKey(node)) {
            return map.get(node);
        }

        // Create a clone of the current node
        UndirectedGraphNode clone = new UndirectedGraphNode(node.label);
        // Add the original node and its clone to the map
        map.put(node, clone);

        // Iterate through the neighbors of the current node
        for (UndirectedGraphNode neighbor : node.neighbors) {
            // Recursively clone each neighbor and add it to the clone's neighbors list
            clone.neighbors.add(auxCloneGraph(neighbor));
        }

        // Return the cloned node
        return clone;
    }
}
```

Black Shapes

Problem Description

Given character matrix **A** of dimensions **N×M** consisting of O's and X's, where O = white, X = black.

Return the number of black shapes. A black shape consists of one or more adjacent X's (diagonals not included)

Problem Constraints

1 <= **N**, **M** <= 1000

A[i][j] = 'X' or 'O'

Input Format

The First and only argument is character matrix **A**.

Output Format

Return a single integer denoting number of black shapes.

Example Input

Input 1:

```
A = [ [X, X, X], [X, X, X], [X, X, X] ]
```

Input 2:

```
A = [ [X, O], [O, X] ]
```

Example Output

Output 1:

```
1
```

Output 2:

```
2
```

Example Explanation

Explanation 1:

```
All X's belong to single shapes
```

Explanation 2:

```
Both X's belong to different shapes
```

```
public class Solution {

    // Directions for exploring neighbors (up, down, left, right)
    private static final int[] rowDir = {-1, 1, 0, 0};
    private static final int[] colDir = {0, 0, -1, 1};

    public int black(String[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        int m = A[0].length();

        // Visited array to keep track of visited cells
        boolean[][] visited = new boolean[n][m];

        // Initialize number of black shapes
        int count = 0;

        // Traverse each cell in the matrix
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                // If current cell is 'X' and not visited yet
```

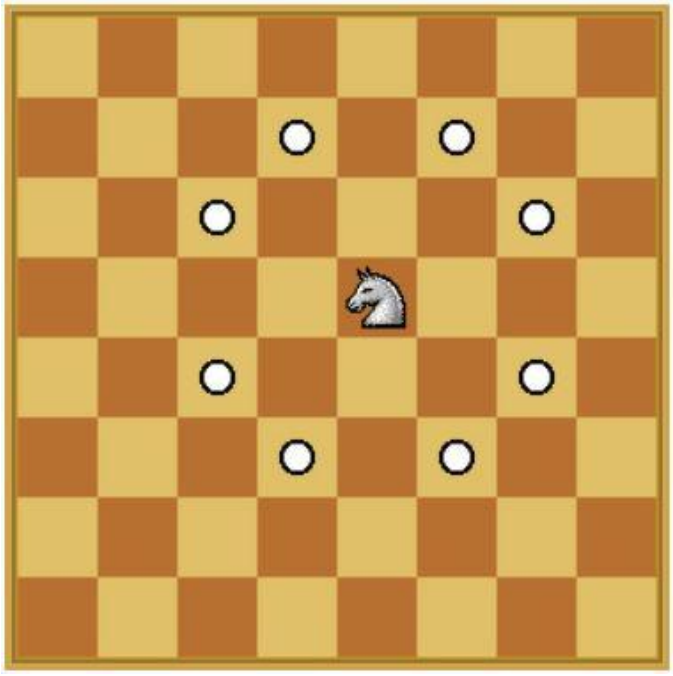
```
            if (A[i].charAt(j) == 'X' && !visited[i][j]) {
                // Perform DFS to mark all connected 'X' cells
                dfs(A, visited, i, j);
                // Increment the count of black shapes
                count++;
            }
        }
    }

    return count;
}

// Depth-First Search (DFS) function to mark all connected 'X' cells
private void dfs(String[] A, boolean[][] visited, int x, int y) {
    // Mark current cell as visited
    visited[x][y] = true;

    // Check all 4 directions (up, down, left, right)
    for (int k = 0; k < 4; k++) {
        int newX = x + rowDir[k];
        int newY = y + colDir[k];

        // Check if the new cell is within bounds and is 'X' and not visited
        if (isValid(A, visited, newX, newY)) {
            dfs(A, visited, newX, newY);
        }
    }
}

// Function to check if a cell is valid and not visited
private boolean isValid(String[] A, boolean[][] visited, int x, int y) {
    int n = A.length;
    int m = A[0].length();
```

```
        return x >= 0 && x < n && y >= 0 && y < m && A[x].charAt(y) == 'X' &&
!visited[x][y];
    }
}
```

Knight On Chess Board

Problem Description

Given any source point, **(C, D)** and destination point, **(E, F)** on a chess board of size **A x B**, we need to find whether Knight can move to the destination or not.



The above figure details the movements for a knight (8 possibilities).

If yes, then what would be the **minimum** number of steps for the knight to move to the said point. If knight can not move from the source point to the destination point, then return **-1**.

NOTE: A knight cannot go out of the board.

Problem Constraints

1 <= A, B <= 500

Input Format

The first argument of input contains an integer A.
The second argument of input contains an integer B.
The third argument of input contains an integer C.
The fourth argument of input contains an integer D.

The fifth argument of input contains an integer E.
The sixth argument of input contains an integer F.

Output Format

If it is possible to reach the destination point, return the minimum number of moves.
Else return -1.

Example Input

Input 1:

```
A = 8
 B = 8
 C = 1
 D = 1
 E = 8
 F = 8
```

Input 2:

```
A = 2
 B = 4
 C = 2
 D = 1
 E = 4
 F = 4
```

Example Output

Output 1:

```
6
```

Output 2:

```
-1
```

Example Explanation

Explanation 1:

```
The size of the chessboard is 8x8, the knight is initially at (1, 1) and the
knight wants to reach position (8, 8).
 The minimum number of moves required for this is 6.
```

Explanation 2:

```
It is not possible to move knight to position (4, 4) from (2, 1)
```

```
import java.util.*;

public class Solution {

    static class Point {
        int x, y, dist;

        public Point(int x, int y, int dist) {
            this.x = x;
            this.y = y;
            this.dist = dist;
        }
    }

    public int knight(int A, int B, int C, int D, int E, int F) {
        // Directions for knight moves
        int[] dx = {-2, -1, 1, 2, 2, 1, -1, -2};
        int[] dy = {1, 2, 2, 1, -1, -2, -2, -1};

        // BFS queue
        Queue<Point> queue = new LinkedList<>();
        queue.offer(new Point(C - 1, D - 1, 0)); // Starting from (C, D)

        // Visited matrix
        boolean[][] visited = new boolean[A][B];
        visited[C - 1][D - 1] = true;
```

```
        // BFS loop
        while (!queue.isEmpty()) {
            Point curr = queue.poll();

            // If we reach the destination
            if (curr.x == E - 1 && curr.y == F - 1) {
                return curr.dist;
            }

            // Explore all 8 possible moves
            for (int i = 0; i < 8; i++) {
                int nx = curr.x + dx[i];
                int ny = curr.y + dy[i];

                // Check if the new position is within bounds and not visited
                if (isValid(nx, ny, A, B) && !visited[nx][ny]) {
                    visited[nx][ny] = true;
                    queue.offer(new Point(nx, ny, curr.dist + 1));
                }
            }
        }

        // If no path found
        return -1;
    }

    // Function to check if a position is valid
    private boolean isValid(int x, int y, int A, int B) {
        return x >= 0 && x < A && y >= 0 && y < B;
    }
}
```

Valid Path

Problem Description

There is a rectangle with left bottom as (0, 0) and right up as (x, y).

There are **N** circles such that their centers are inside the rectangle.

Radius of each circle is **R**. Now we need to find out if it is possible that we can move from (0, 0) to (x, y) without touching any circle.

Note : We can move from any cell to any of its 8 adjecent neighbours and we cannot move outside the boundary of the rectangle at any point of time.

Problem Constraints

0 <= **x** , **y**, **R** <= 100

1 <= **N** <= 1000

Center of each circle would lie within the grid

Input Format

1st argument given is an Integer **x** , denoted by A in input.

2nd argument given is an Integer **y**, denoted by B in input.

3rd argument given is an Integer **N**, number of circles, denoted by C in input.

4th argument given is an Integer **R**, radius of each circle, denoted by D in input.

5th argument given is an Array **A** of size N, denoted by E in input, where A[i] = x cordinate of ith circle

6th argument given is an Array **B** of size N, denoted by F in input, where B[i] = y cordinate of ith circle

Output Format

Return YES or NO depending on weather it is possible to reach cell (x,y) or not starting from (0,0).

Example Input

Input 1:

```
x = 2
 y = 3
 N = 1
 R = 1
 A = [2]
 B = [3]
```

Input 2:

```
x = 3
 y = 3
 N = 1
 R = 1
 A = [0]
 B = [3]
```

Example Output

Output 1:

```
NO
```

Output 2:

```
YES
```

Example Explanation

Explanation 1:

```
There is NO valid path in this case
```

Explanation 2:

```
There is many valid paths in this case.
 One of the path is (0, 0) -> (1, 0) -> (2, 0) -> (3, 0) -> (3, 1) -> (3, 2) ->
(3, 3).
```

```
import java.util.*;

public class Solution {

    // Directions for moving in the grid (8 possible moves)
    static final int[] dx = {1, 1, 0, -1, -1, -1, 0, 1};
    static final int[] dy = {0, 1, 1, 1, 0, -1, -1, -1};

    public String solve(int A, int B, int N, int R, int[] C, int[] D) {
        // Create the grid and mark cells that are within R distance of any circle as true
(occupied)
        boolean[][] grid = new boolean[A + 1][B + 1];

        for (int i = 0; i <= A; i++) {
            for (int j = 0; j <= B; j++) {
                for (int k = 0; k < N; k++) {
                    int circleX = C[k];
                    int circleY = D[k];

                    if (isInsideCircle(i, j, circleX, circleY, R)) {
                        grid[i][j] = true;
                        break; // No need to check further circles for this cell
                    }
                }
            }
        }

        // Use BFS to check if we can reach (A, B) from (0, 0)
        if (canReachDestination(grid, A, B)) {
            return "YES";
```

```
    } else {
        return "NO";
    }
}

// Helper function to check if a cell is inside the radius of a circle
private boolean isInsideCircle(int x, int y, int cx, int cy, int R) {
    int dx = x - cx;
    int dy = y - cy;
    return dx * dx + dy * dy <= R * R;
}

// Helper function to perform BFS to check if we can reach (A, B) from (0, 0)
private boolean canReachDestination(boolean[][] grid, int A, int B) {
    Queue<int[]> queue = new LinkedList<>();
    boolean[][] visited = new boolean[A + 1][B + 1];

    queue.offer(new int[]{0, 0});
    visited[0][0] = true;

    while (!queue.isEmpty()) {
        int[] curr = queue.poll();
        int x = curr[0];
        int y = curr[1];

        // If we reached the destination
        if (x == A && y == B) {
            return true;
        }

        // Explore 8 possible moves
        for (int i = 0; i < 8; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];
```

```
            // Check if the move is within bounds and not visited and not occupied by a
circle
            if (nx >= 0 && nx <= A && ny >= 0 && ny <= B && !visited[nx][ny] &&
!grid[nx][ny]) {
                visited[nx][ny] = true;
                queue.offer(new int[]{nx, ny});
            }
        }
    }

    // If we exhausted all possibilities and did not reach (A, B)
    return false;
  }
}
```

Distance of nearest cell

Problem Description

Given a matrix of integers **A** of size **N x M** consisting of **0** or **1**.

For each cell of the matrix find the distance of nearest 1 in the matrix.

Distance between two cells **(x1, y1)** and **(x2, y2)** is defined as **|x1 - x2| + |y1 - y2|**.

Find and return a matrix **B** of size **N x M** which defines for each cell in A distance of nearest **1** in the matrix A.

NOTE: There is atleast one 1 is present in the matrix.

Problem Constraints

1 <= N, M <= 1000

0 <= A[i][j] <= 1

Input Format

The first argument given is the integer matrix A.

Output Format

Return the matrix B.

Example Input

Input 1:

```
A = [
        [0, 0, 0, 1]
        [0, 0, 1, 1]
        [0, 1, 1, 0]
     ]
```

Input 2:

```
A = [
        [1, 0, 0]
        [0, 0, 0]
        [0, 0, 0]
     ]
```

Example Output

Output 1:

```
[
   [3, 2, 1, 0]
   [2, 1, 0, 0]
   [1, 0, 0, 1]
 ]
```

Output 2:

```
[
   [0, 1, 2]
   [1, 2, 3]
   [2, 3, 4]
 ]
```

Example Explanation

Explanation 1:

```
A[0][0], A[0][1], A[0][2] will be nearest to A[0][3].
 A[1][0], A[1][1] will be nearest to A[1][2].
 A[2][0] will be nearest to A[2][1] and A[2][3] will be nearest to A[2][2].
```

Explanation 2:

```
There is only a single 1. Fill the distance from that 1.
```

```
import java.util.*;

public class Solution {
    public int[][] solve(int[][] A) {
        int N = A.length;
        int M = A[0].length;

        // Output matrix to store distances
        int[][] B = new int[N][M];
        // Queue for BFS
        Queue<int[]> queue = new LinkedList<>();
        // Visited matrix to mark visited cells
        boolean[][] visited = new boolean[N][M];

        // Initialize queue and visited matrix with all cells containing 1
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                if (A[i][j] == 1) {
                    queue.offer(new int[]{i, j});
                    visited[i][j] = true;
                    B[i][j] = 0;
                }
            }
```

```
        }

        // Directions for moving in 4 possible directions (up, down, left, right)
        int[] dx = {-1, 1, 0, 0};
        int[] dy = {0, 0, -1, 1};

        // Perform BFS
        while (!queue.isEmpty()) {
            int[] curr = queue.poll();
            int x = curr[0];
            int y = curr[1];

            // Explore neighbors
            for (int k = 0; k < 4; k++) {
                int nx = x + dx[k];
                int ny = y + dy[k];

                // Check if the neighbor is within bounds and not visited
                if (nx >= 0 && nx < N && ny >= 0 && ny < M && !visited[nx][ny]) {
                    visited[nx][ny] = true;
                    B[nx][ny] = B[x][y] + 1;
                    queue.offer(new int[]{nx, ny});
                }
            }
        }

        return B;
    }
}
```

Number of islands II

Problem Description

Given a matrix of integers **A** of size **N x M** consisting of **0** and **1**. A group of connected 1's forms an island. From a cell `(i, j)` such that `A[i][j] = 1` you can visit any cell that shares a side with `(i, j)` and value in that cell is **1**.

More formally, from any cell `(i, j)` if `A[i][j] = 1` you can visit:

`(i-1, j)` if `(i-1, j)` is inside the matrix and `A[i-1][j] = 1`.

`(i, j-1)` if `(i, j-1)` is inside the matrix and `A[i][j-1] = 1.`

`(i+1, j)` if `(i+1, j)` is inside the matrix and `A[i+1][j] = 1.`

`(i, j+1)` if `(i, j+1)` is inside the matrix and `A[i][j+1] = 1.`

Return the **number of islands**.

Note:

Rows are numbered from **top to bottom** and columns are numbered from **left to right**.
Your solution will run on multiple test cases. If you are using global variables, make sure to clear them.

Problem Constraints

1 <= N, M <= 100
0 <= A[i] <= 1

Input Format

The only argument given is the integer matrix A.

Output Format

Return the number of islands.

Example Input

Input 1:

```
A = [ [0, 1, 0]
      [0, 0, 1]
      [1, 0, 0] ]
```

Input 2:

```
A = [ [1, 1, 0, 0, 0]
      [1, 1, 0, 0, 0]
      [0, 0, 0, 0, 0]
      [0, 0, 0, 1, 1] ]
```

Example Output

Output 1:

```
3
```

Output 2:

```
2
```

Example Explanation

Explanation 1:

```
There are 3 islands in the matrix
```

Explanation 2:

```
There are 2 islands in the matrix
```

```
public class Solution {
    // Directions for 4 possible moves (up, down, left, right)
    private static final int[] dx = {-1, 1, 0, 0};
    private static final int[] dy = {0, 0, -1, 1};

    public int numIslands(int[][] A) {
        int N = A.length;
        int M = A[0].length;
        boolean[][] visited = new boolean[N][M];
        int islandsCount = 0;
```

```
        // Traverse each cell in the matrix
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                // If land (A[i][j] == 1) and not visited
                if (A[i][j] == 1 && !visited[i][j]) {
                    islandsCount++; // Found a new island
                    dfs(A, visited, i, j); // Visit all connected land cells
                }
            }
        }

        return islandsCount;
    }

    // DFS to mark all connected land cells as visited
    private void dfs(int[][] A, boolean[][] visited, int x, int y) {
        int N = A.length;
        int M = A[0].length;

        // Mark current cell as visited
        visited[x][y] = true;

        // Explore 4 possible directions
        for (int k = 0; k < 4; k++) {
            int nx = x + dx[k];
            int ny = y + dy[k];

            // Check if the next cell is within bounds and is land
            if (nx >= 0 && nx < N && ny >= 0 && ny < M && A[nx][ny] == 1 &&
!visited[nx][ny]) {
                dfs(A, visited, nx, ny); // Recursively visit the connected land cells
            }
        }
```

```
    }
}
```